Optimal Construction of Hierarchical Overlap Graphs

Khan, Shahbaz

Schloss Dagstuhl - Leibniz-Zentrum für Informatik
2021-07-06

Khan , S 2021 , Optimal Construction of Hierarchical Overlap Graphs . in P Gawrychowski & T Starikovskaya (eds) , 32nd Annual Symposium on Combinatorial Pattern Matching : CPM 2021 . Leibniz International Proceedings in Informatics (LIPIcs) , vol. 191 , Schloss Dagstuhl - Leibniz-Zentrum für Informatik , Germany , pp. 17:1-17:11 , Annual Symposium on þÿ C o m b i n a t o r i a l P a t t e r n M a t c h i n g , W r o c B a w , P o l a n d , 0 5 / 0 7 / 2 0 2 1 . h t t p

http://hdl.handle.net/10138/334910
https://doi.org/10.4230/LIPIcs.CPM.2021.17

Optimal Construction of Hierarchical Overlap Graphs

Shahbaz Khan
University of Helsinki, Finland

Abstract

Genome assembly is a fundamental problem in Bioinformatics, where for a given set of overlapping substrings of a genome, the aim is to reconstruct the source genome. The classical approaches to solving this problem use assembly graphs, such as *de Bruijn graphs* or *overlap graphs*, which maintain partial information about such overlaps. For genome assembly algorithms, these graphs present a trade-off between overlap information stored and scalability. Thus, Hierarchical Overlap Graph (HOG) was proposed to overcome the limitations of both these approaches.

For a given set P of n strings, the first algorithm to compute HOG was given by Cazaux and Rivals [IPL20] requiring $O(||P|| + n^2)$ time using superlinear space, where $||P||$ is the cumulative sum of the lengths of strings in P. This was improved by Park et al. [SPIRE20] to $O(||P|| \log n)$ time and $O(||P||)$ space using segment trees, and further to $O(||P|| \frac{\log n}{\log \log n})$ for the word RAM model. Both these results described an open problem to compute HOG in optimal $O(||P||)$ time and space. In this paper, we achieve the desired optimal bounds by presenting a simple algorithm that does not use any complex data structures. At its core, our solution improves the classical result [IPL92] for a special case of the All Pairs Suffix Prefix (APSP) problem from $O(||P|| + n^2)$ time to optimal $O(||P||)$ time, which may be of independent interest.

2012 ACM Subject Classification Mathematics of computing → Trees; Theory of computation → Data compression; Theory of computation → Pattern matching

Keywords and phrases Hierarchical Overlap Graphs, String algorithms, Genome assembly

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.17

Related Version *Previous Version*: `https://arxiv.org/abs/2102.02873` [14]

Funding This work was funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO).

Acknowledgements I would like to thank Alexandru I. Tomescu for helpful discussions, and for critical review and insightful suggestions which helped me in refining the paper. I would also like to thank Veli Mäkinen for pointing out the similarity with the classical result for APSP problem.

1 Introduction

Genome assembly is one of the oldest and most fundamental problems in Bioinformatics [21]. Due to practical limitations, sequencing an entire genome as a single complete string is not possible, rather a collection of the *substrings* of the genome (called *reads*) are sequenced. The goal of a sequencing technology is to produce a collection of reads that cover the entire genome and have sufficient overlap amongst the reads. This allows the source genome to be reconstructed by ordering the reads using this overlap information. The genome assembly problem thus aims at computing the source genome given such a collection of overlapping reads. Most approaches of genome assembly capture this overlap information into an *assembly graph*, which can then be efficiently processed to assemble the genome. The prominent approaches use assembly graphs such as *de Bruijn graphs* [22] and *Overlap graphs* (also called string graphs [17]), which have been shown to be successfully used in various practical assemblers [28, 3, 18, 2, 23, 24].

The de Bruijn graphs are built over k length substrings (or k-mers) of the reads as nodes, and arcs denoting $k-1$ length overlaps among the k-mers. Their prominent advantage is that their size is linear in that of the input. However, their limitations include losing information about the relationship of k-mers with the reads, and in general not being able to represent overlaps of size other than $k-1$ among the reads (except [7, 5, 4]). On the other hand, Overlap graphs have each read as a node, and edges between every pair of nodes represent their corresponding maximum overlap. In practice, only the edges having certain threshold value of overlap are considered. Though they store more overlap information than de Bruijn graphs, they do not maintain whether two pairs of strings have the same overlap. Moreover, they are inherently quadratic in size in the worst case, and computing the edge weights (even optimally [13, 26, 16]) is difficult in practice for large data sets.

As a result, Hierarchical Overlap Graphs (HOG) were proposed in [9, 10] as an alternative to overcome such limitations of the two types of assembly graphs. The HOG has nodes for all the longest overlaps between every pair of strings, and edges connecting strings to their suffix and prefix, using linear space. Note that Overlap graphs have edges representing longest overlaps between strings requiring quadratic size, whereas HOG has additional nodes for longest overlaps between strings requiring linear size by exploiting pairs of strings having the same longest overlaps. Thus, it is a promising alternative to both de Bruijn graph and Overlap graph to better solve the problem of genome assembly. Also, since it maintains if two pairs of strings have the same overlap, it also has the potential to better solve the approximate *shortest superstring problem* [27] having applications in both genome assembly and data compression [25, 6]. Some applications of HOG have been studied in [9, 8].

Cazaux and Rivals [10] presented the first algorithm to build HOG efficiently. They showed how HOG can be computed for a set of n strings P in $O(||P|| + n^2)$ time, where $||P||$ represents the cumulative sum of lengths of strings in P. However, they required $O(||P|| + n \times \min(n, \max_{p \in P} |p|))$ space, which is superlinear in input size. Park et al. [20] improved it to $O(||P|| \log n)$ time requiring linear space using Segment trees [11], assuming a constant sized character set. For the word RAM model, they further improved it to $O(||P|| \frac{\log n}{\log \log n})$ time. For practical implementation, both these results build HOG using an intermediate Extended HOG (EHOG) which reduces the memory footprint of the algorithm. In both the results, the *bottleneck* is solving a special case of All Pairs Suffix Prefix (APSP) problem. Given a set P of n strings, the goal of the APSP problem is to compute the maximum overlaps between every pair of strings. This classical problem was optimally solved by Gusfield et al. [13] using $O(||P|| + n^2)$ time and $O(||P||)$ space, where the solution is reported for the n^2 pairs. However, for computing HOG we only require the set of *maximum overlaps*, and not their association with the corresponding pairs of strings, making the result suboptimal due to the extra $O(n^2)$ factor. Also, both these results [10, 20] mentioned as an open problem the construction of HOG using optimal $O(||P||)$ time and space. We answer this open question positively and solve the special case of APSP optimally as follows.

▶ **Theorem 1** (Optimal HOG). *For a set of strings P, the Hierarchical Overlap Graph can be computed using $O(||P||)$ time and space.*

Moreover, unlike [20] our algorithm does not use any complex data structures for its implementation. Also, we do not assume any limitations on the character set. Finally, like [10, 20] our algorithm can also use EHOG as an intermediate step for improving memory footprint in practice. Note that the size EHOG and HOG can even be identical for some instances, but their ratio can tend to infinity for some families of graphs [10]. Thus, despite the existence of optimal algorithm for computing EHOG, an optimal algorithm for computing HOG is significant from both theoretical and practical viewpoints.

Note. Another result [19] simultaneously achieve the same optimal bound by reducing the problem to computing *borders* [15]. However, our result is simpler and more self-contained.

Outline of the paper. We first describe notations and preliminaries that are used in our paper in Section 2. In Section 3, we briefly describe the previous approaches to compute HOG. Thereafter, Section 4 describes our core result in three stages for simplicity of understanding, each building over the previous, to give the optimal algorithm. Finally, we present the conclusions in Section 5.

2 Preliminaries

Given a finite set $P = \{p_1, ..., p_n\}$ of n non-empty strings over a finite set of characters, we denote the size of a string p_i by $|p_i|$ and the cumulative size of P by $||P|| = \sum_{i=1}^{n} |p_i|$ ($\geq n$ as strings are not empty). For a string p, any substring that starts from the first character of p is called a *prefix* of p, whereas any substring which ends at the last character of p is called a *suffix* of p. A prefix or suffix of p is called *proper* if it is not same as the whole p. For an ordered pair of string (p_1, p_2), a string is called their *overlap* if it is both a proper suffix of p_1 and a proper prefix of p_2, where $ov(p_1, p_2)$ denotes the *longest* such overlap. Also, for the set of strings P, $Ov(P)$ denotes the set of all $ov(p_i, p_j)$ for $1 \leq i, j \leq n$. An empty string is denoted by ϵ. We also use the notions of HOG, EHOG and the Aho-Corasick trie as follows.

▶ **Definition 2** (Hierarchical Overlap Graph [10]). *Given a set of strings $P = \{p_1, \cdots, p_n\}$, its Hierarchical Overlap Graph is a directed graph $\mathcal{H} = (V, E)$, where*

- *$V = P \cup Ov(P) \cup \{\epsilon\}$ and $E = E_1 \cup E_2$, having*
- *$E_1 = \{(x, y) : x$ is the longest proper prefix of y in $V\}$ as* tree edges*, and*
- *$E_2 = \{(x, y) : y$ is the longest proper suffix of x in $V\}$ as* suffix links*.*

The *extended HOG* of P (referred as $\mathcal{E}$) is also similarly defined [10], having additional nodes corresponding to every overlap (not just longest) between each pair of strings in P, with the same definition of edges. The construction of both these structures uses the Aho-Corasick Trie [1] which is computable in $O(||P||)$ time and space. The Aho-Corasick Trie of P (referred as $\mathcal{A}$) contains all prefixes of strings in P as nodes, with the same definition for edges. All these structures are essentially *trees* having the empty string ϵ as the root, and the strings of P as its *leaves*. A *tree edge* (x, y) is labelled with the substring of y not present in x. Hence, despite being a graph due to the presence of *suffix links* (also called *failure links*), we abuse the notions used for tree structures when applying to $\mathcal{A}, \mathcal{E}$ or $\mathcal{H}$ (ignoring suffix links). Also, while referring to a node v of $\mathcal{A}, \mathcal{E}$ or $\mathcal{H}$, we represent its corresponding string with v as well.

Consider Figure 1 for a comparison of $\mathcal{A}$, $\mathcal{E}$ and $\mathcal{H}$ for $P = \{aabaa, aadbd, dbdaa\}$. Since $\mathcal{A}$ contains all prefixes as nodes, the tree edges have labels of a single character. However, $\mathcal{E}$ contains all overlaps among strings of P, so it can potentially have fewer internal nodes ($\{a, aa, db, dbd\}$) than $\mathcal{A}$. Further, $\mathcal{H}$ contains only longest overlaps so it can potentially have even fewer internal nodes ($\{aa, dbd\}$).

Now, to compute $\mathcal{E}$ or $\mathcal{H}$ one must only remove some internal nodes from $\mathcal{A}$ and adjust the edge labels accordingly. This requires the computation of all overlaps among strings in P for $\mathcal{E}$, which is further restricted to only the longest overlaps for $\mathcal{H}$. For a string $p_i \in P$ (leaf of $\mathcal{A}$), all its prefixes are its ancestors in $\mathcal{A}$, whereas all its suffixes are on the path following the suffix links from it (referred as *suffix path*). Thus, every internal node is implicitly the prefix of its descendant leaves, and to be an overlap it must merely be a suffix of some string

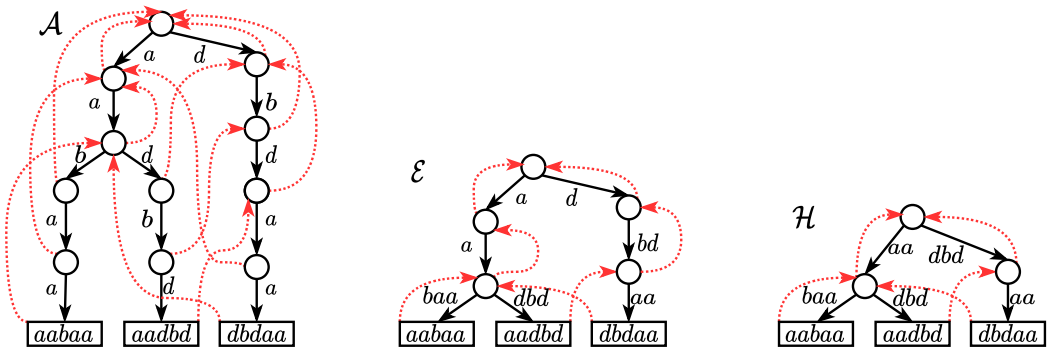

Figure 1 Given $P = \{aabaa, aadbd, dbdaa\}$, the figure shows from left to right the Aho-Corasick Trie ($\mathcal{A}$), Extended Hierarchical Overlap Graph ($\mathcal{E}$) and Hierarchical Overlap Graph ($\mathcal{H}$) of P.

in P [27]. Hence to compute internal nodes of $\mathcal{E}$ (or overlap) from $\mathcal{A}$ one simply traverses the suffix paths from all the leaves of $\mathcal{A}$, and remove the non-traversed internal nodes (see Figure 1). However, to compute $\mathcal{H}$ from $\mathcal{A}$ (or $\mathcal{E}$) we need to find only the longest overlaps, which is equivalent to solving a special case of the APSP problem, requiring only the set of all maximum overlaps. We use the following criterion (also used by [13]) to identify the internal nodes of $\mathcal{H}$.

▶ **Lemma 3** ([13]). *An internal node v in $\mathcal{A}$ (or $\mathcal{E}$) of P, is $ov(p_i, p_j)$ for two strings $p_i, p_j \in P$ iff v is an overlap of (p_i, p_j) and no descendant of v is an overlap of (p_i, p_j).*

Proof. The ancestor of a node v in $\mathcal{A}$ is its proper prefix and hence is shorter than v. Since two internal nodes of $\mathcal{A}$ which are both overlaps of (p_i, p_j), are prefixes of p_j and hence have an ancestor-descendant relationship, where the descendant is longer in length. Thus, the longest overlap $ov(p_i, p_j)$ cannot have a descendant which is an overlap of (p_i, p_j). ◀

Hence to compute $Ov(P)$ (or nodes of $\mathcal{H}$), we need to check each internal node v if it is the lowest overlap (in $\mathcal{A}$) for some pair (p_i, p_j). This implies that v is a suffix of some p_i, such that for some descendant leaf p_j, no suffix of p_i is on path from v to p_j (see Figure 1).

3 Previous results

Cazaux and Rivals [10] were the first to study $\mathcal{H}$, where they used $\mathcal{E}$ [8] as an intermediate step in the computation of $\mathcal{H}$. They showed that $\mathcal{E}$ can be constructed in $O(||P||)$ time and space from $\mathcal{A}$ [1], which itself is computable in $O(||P||)$ time and space. In order to compute $\mathcal{H}$, the main *bottleneck* is the computation of $Ov(P)$ (i.e. solving APSP), after which we simply remove the internal nodes not in $Ov(P)$ from $\mathcal{E}$ (or $\mathcal{A}$), in $O(||P||)$ time and space. They gave an algorithm to compute $Ov(P)$ in $O(||P|| + n^2)$ time using $O(||P|| + n \times \min(n, \max\{|p_i|\}))$ space. This procedure was recently improved by Park et al. [20] to require $O(||P|| \log n)$ time and $O(||P||)$ space using segment trees, assuming constant sized character set. For the word RAM model they further improve the time to $O(||P|| \frac{\log n}{\log \log n})$. The main ideas of the previous results can be summarized as follows.

Computing $Ov(P)$ in $O(||P|| + n^2)$ time [10]

The algorithm computes $Ov(P)$ by considering the internal nodes in a bottom-up manner, where a node is processed after its descendants. Firstly, for each internal node u, they compute the list $R_l(u)$ (called $\mathcal{L}_u$ in our algorithm) of all leaves having u as a suffix. Now,

while processing a node u, they check whether $u = ov(v, x)$, i.e., u is a suffix of some leaf v such that the path to at least one of u's descendant leaf (say x) does not have a suffix of v. To perform this task, they maintain a bit-vector for all leaves (suffix v), which is marked if no such descendant path exists from u for such leaves. For a leaf v, the bit is implicitly marked if all children of u have the bit for v marked. Otherwise, if $v \in R_l(u)$ it is marked adding u to $\mathcal{H}$, else left unmarked. The space requirement is dominated by that of this bit-vector, and it is computed only for the branching nodes, taking total $O(||P|| + n^2)$ time.

Computing $Ov(P)$ in $O(||P|| \log n)$ time [20]

The algorithm firstly orders the strings in P lexicographically in $O(||P||)$ time (requires constant sized character set). This allows them to define an interval of leaves which are the descendants of each internal node in $\mathcal{E}$. Now, for each leaf v (suffix) they start with an unmarked array corresponding to all leaves (prefix). Then starting from v they follow its suffix path and at each internal node u, check if some descendant leaf x (prefix) is unmarked. In such a case $u = ov(v, x)$ and hence u is added to $\mathcal{H}$. Before moving further in the next suffix path the interval corresponding to all the descendant leaves (prefix) of u is marked in the array. Since both query and update (mark) over an interval can be performed in $O(\log n)$ time using a segment tree, the total time taken is $O(||P|| \log n)$ using $O(||P||)$ space.

4 Our algorithm

Our main contribution is an alternative procedure to compute $Ov(P)$ in $O(||P||)$ time and space which results in an optimal algorithm for computing $\mathcal{H}$ for P in $O(||P||)$ time and space. Our overall approach is similar to that of the original algorithm [10] with the exception of a procedure to mark the internal nodes that belong to $\mathcal{H}$, i.e., Mark$\mathcal{H}$. The algorithm except for the procedure Mark$\mathcal{H}$ takes $O(||P||)$ time and space (also shown in [10]). We describe our algorithm for Mark$\mathcal{H}$ in three stages, first for a single prefix leaf requiring $O(||P||)$ time, and then for all prefix leaves requiring overall $O(||P|| + n^2)$ time, and finally improving it to overall $O(||P||)$ time, which is optimal. The algorithm can be applied to any of $\mathcal{A}$ or $\mathcal{E}$, both computable in $O(||P||)$ time and space.

Note: The second stage of our algorithm is equivalent to [13], and achieves the same bounds as [10] for computing $\mathcal{H}$, though using a simpler technique and linear space.

4.1 Outline of Approach

We first describe our overall approach in Algorithm 1. After computing $\mathcal{A}$, for each internal node v, we compute the list $\mathcal{L}_v$ of all the leaves having v as its suffix. As described earlier, this can be done by following the suffix path of each leaf x, adding x to $\mathcal{L}_y$ for every internal node y on the path. Using this information of suffix (in $\mathcal{L}_v$) and prefix (implicit in $\mathcal{A}$) we mark the nodes of $\mathcal{A}$ to be added in the HOG $\mathcal{H}$. We shall describe this procedure Mark$\mathcal{H}$ later on. Thereafter, in order to compute $\mathcal{H}$ we simply merge the unmarked internal nodes of $\mathcal{A}$ with its parents. This process is carried on using a DFS traversal of $\mathcal{A}$ (ignoring suffix links) where for each unmarked internal node v, we move all its edges to its parent, prepending their labels with the label of the parent edge of v.

As previously described, $\mathcal{A}$ can be computed in $O(||P||)$ time and space [1]. Computing $\mathcal{L}_v$ for all $v \in \mathcal{A}$ requires each leaf p_i to follow its suffix path in $O(|p_i|)$ time, and add p_i to at most $|p_i|$ different $\mathcal{L}_y$, requiring total $O(||P||)$ time for all $p_i \in P$. This also limits the

Algorithm 1 HIERARCHICAL OVERLAP GRAPHS.

$\mathcal{A} \leftarrow$ Aho-Corasik Trie of P `// Trie with suffix links`

foreach *internal node* v *of* $\mathcal{A}$ **do** $\mathcal{L}_v \leftarrow \emptyset$ `// List of leaves with suffix` v

foreach *leaf* x *of* $\mathcal{A}$ **do** `// Compute all` $\mathcal{L}_v$

  $y \leftarrow$ Suffix link of x in $\mathcal{A}$

  while $y \neq \epsilon$ **do** `//` ϵ `is the root of` $\mathcal{A}$

    Add x to $\mathcal{L}_y$

    $y \leftarrow$ Suffix link of y in $\mathcal{A}$

$in\mathcal{H} \leftarrow$Mark$\mathcal{H}(\mathcal{A}, \mathcal{L})$ `// Procedure to mark nodes of` $\mathcal{H}$ `in flags` $in\mathcal{H}$

foreach *node* $v \in \mathcal{A}$ *in DFS order* **do** `// Compute` $\mathcal{H}$

  if $in\mathcal{H}[v] = 0$ **then** Merge v with its parent

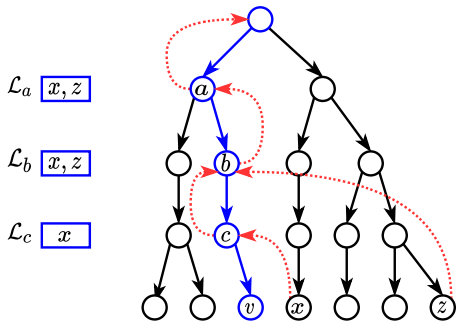

Figure 2 Overlaps of v with all leaves, where $c = ov(x, v)$ and $b = ov(z, v)$ are in $Ov(P)$.

size of $\mathcal{L}_v$ for all $v \in \mathcal{A}$ to $O(||P||)$. Since merge operation on a node v requires $O(deg(v))$ cost, computing $\mathcal{H}$ using $in\mathcal{H}$ requires total $O(|\mathcal{A}|) = \mathcal{O}(||\mathcal{P}||)$ time as well. Thus, we have the following theorem (also proved in [10]).

▶ **Theorem 4.** *For a set of strings P, the computation of Hierarchical Overlap Graph except for Mark$\mathcal{H}$ operation requires $O(||P||)$ time and space.*

4.2 Marking the nodes of $\mathcal{H}$

We shall describe our procedure to mark the nodes of $\mathcal{H}$ in three stages for simplicity of understanding. First, we shall describe how to mark all internal nodes representing all longest overlaps $ov(\cdot, v)$ from a single leaf v (prefix) in $\mathcal{A}$, using $O(||P||)$ time. Thereafter, we extend this to compute such overlaps from all leaves in $\mathcal{A}$ together using $O(||P|| + n^2)$ time (equivalent to [13]). Finally, we shall improve this to our final procedure requiring optimal $O(||P||)$ time. All the three procedures require $O(||P||)$ space.

Marking all nodes $ov(\cdot, v)$ for a leaf v

In order to compute all longest overlaps of a leaf v (see Figure 2), we need to consider all its prefixes (ancestors in $\mathcal{A}$) according to Lemma 3. Here the internal nodes a, b and c are prefixes of v and also suffixes of x, whereas z only has suffixes a and b. Thus, we have $\mathcal{L}_a = \mathcal{L}_b = \{x, z\}$ and $\mathcal{L}_c = \{x\}$. Thus, given that a, b and c are ancestors of v, a and b are

valid overlaps of (x, v) and (z, v), whereas c is only a valid overlap of (x, v). Using Lemma 3, for being the longest overlap of a pair of strings, no descendant should be an overlap of the same pair of strings. Hence, $c = ov(x, v)$ and $b = ov(z, v)$, but a is not the longest overlap for any pair of strings because of b and c. Processing $\mathcal{L}_u$ for all nodes on the ancestors of the leaf (prefix) requires $O(||P||)$ time. Thus, a simple way to mark all the longest overlaps of strings with prefix v in $O(||P||)$ time, is as follows:

Mark$\mathcal{H}$ for $ov(\cdot, v)$:
Traverse the ancestral path of v from the root to v, storing for each leaf x of $\mathcal{A}$ the last internal node y having x in $\mathcal{L}_y$. On reaching v, mark the stored internal nodes for each x.

Algorithm 2 MARK$\mathcal{H}(\mathcal{A}, \mathcal{L})$.

```
foreach internal node v of A do inH[v] ← 0      // Flag for membership in H
foreach leaf v of A do inH[v] ← 1               // Leaves implicitly in H
inH[ϵ] ← 1                                      // Root implicitly in H
foreach leaf v of A do S_v ← ∅                  // Stack of exposed suffix
foreach node v ∈ A in DFS order do              // Compute all inH[v]
    if internal node v first visited then
        foreach leaf x in L_v do Push v on S_x  // Expose v on stacks of L_v
    if internal node v last visited then
        foreach leaf x in L_v do Pop v from S_x // Remove v from stacks of L_v
    if leaf v visited then
        foreach leaf node x do
            if S_x ≠ ∅ then
                inH[Top of S_x] ← 1             // Mark ov(x, v)
Return inH
```

Marking all nodes in $Ov(P)$

We now describe how to perform this procedure for all leaves (prefix) together (see Algorithm 2) using stacks to keep track of the last encountered internal node for each leaf (suffix). The main reason behind using stacks is to avoid processing $\mathcal{L}_u$ multiple times (for different prefixes). For each internal node, we initialize the flag denoting membership in $\mathcal{H}$ to zero, whereas the root and leaves of $\mathcal{A}$ are implicitly in $\mathcal{H}$. For each leaf (suffix) we initialize an empty stack. Now, we traverse $\mathcal{A}$ in DFS order (ignoring suffix links). As in the case for single leaf (prefix), the stack S_x maintains the last internal node v containing a leaf x (suffix) in $\mathcal{L}_v$. This node v is added to the stack S_x of the leaf x (suffix) when v is first visited by the traversal, and removed from the stack S_x when it is last visited. This exposes the previously added internal nodes on the stack. Finally, on visiting a leaf v (prefix), each non-empty stack S_x of a leaf x (suffix) exposes the internal node last added on its top, which is the longest overlap $ov(x, v)$ by Lemma 3. We mark such internal nodes as being present in $\mathcal{H}$. The correctness follows from the same arguments used for the first approach.

In order to analyze the procedure we need to consider the processing of $\mathcal{L}_v$ and S_x for all $v, x \in \mathcal{A}$, in addition to traversing $\mathcal{A}$. Since the total size of all $\mathcal{L}_v$ is $O(||P||)$, processing it twice (on the first and last visit of v) requires $O(||P||)$ time. This also includes the time to push and pop nodes from the stacks, requiring $O(1)$ time while processing $\mathcal{L}_v$. However, on visiting the leaf (prefix) by the traversal, we need to evaluate all S_x and mark the top of non-empty stacks. Since we consider n leaves (prefix), each processing all stacks of n leaves (suffix), we require $O(n^2)$ time. For analyzing size, we need to consider only S_x in addition to $\mathcal{L}_v$. Since the nodes in all S_x are added once from some $\mathcal{L}_v$, the total size of all stacks S_x is bounded by the size of all lists $\mathcal{L}_v$, i.e. $O(||P||)$ (as proved earlier). Thus, this procedure requires $O(||P|| + n^2)$ time and $O(||P||)$ space to mark all nodes in $Ov(P)$.

Optimizing Mark$\mathcal{H}$

As described earlier, the only operation not bounded by $O(||P||)$ time is the marking of internal nodes, while processing the leaves (prefix) considering the stacks of all leaves (suffix). Note that this procedure is overkill as the same top of the stack can be marked again when processing different leaves (prefix), whereas total nodes entering and leaving stacks are proportional to total size of all $\mathcal{L}_u$, i.e., $O(||P||)$. Thus, we ensure that we do not have to process stacks of all leaves (suffix) on processing the leaves (prefix) of $\mathcal{A}$, and instead, we only process those stacks which were not processed earlier to mark the same top. Note that the same internal node may be marked again when exposed in different stacks, but we ensure that it is not marked again while processing the same stack.

Consider Algorithm 3 (showing modified code in red and additions in blue), we maintain a doubly linked-list $\mathcal{S}$ of non-empty stacks whose tops are not marked. Now, whenever a new node is added to a stack, it clearly has an unmarked top, so it is added to $\mathcal{S}$. And when a node is removed from a stack, the stack is added to $\mathcal{S}$ if the new top is not previously marked and stack in not already in $\mathcal{S}$. Similarly, if the stack is empty or has a previously marked top, it is removed from $\mathcal{S}$ if it was present in $\mathcal{S}$. Since $\mathcal{S}$ is a list, its members are additionally maintained using flags $in\mathcal{S}$ for each stack corresponding to leaves (suffix) of $\mathcal{A}$, so that the same stack is not added multiple times in $\mathcal{S}$. Also, each stack in $\mathcal{S}$ maintains a pointer to its location in $\mathcal{S}$, so that it can be efficiently removed if required. Now, on processing the leaves (prefix) of $\mathcal{A}$, we only process the stacks in $\mathcal{S}$, marking their tops and removing them from $\mathcal{S}$. Clearly, stacks are added to $\mathcal{S}$ only while processing $\mathcal{L}_v$, hence overall we can mark $O(|\mathcal{L}_v|)$ nodes for all v, requiring total $O(||P||)$ time. And the time taken in removing stacks from $\mathcal{S}$ is bounded by the total size of all S_x, which is also $O(||P||)$. Thus, we can perform Mark$\mathcal{H}$ using optimal $O(||P||)$ time and space, which results in our main result (using Theorem 4).

▶ **Theorem 1** (Optimal HOG). *For a set of strings P, the Hierarchical Overlap Graph can be computed using $O(||P||)$ time and space.*

Remark: The classical result for APSP [13] (equivalent to our second stage) was optimized [12] to get *output-sensitive* $O(||P|| + n')$ time (where n' is number of pairs with non-zero overlap) by maintaining a list of non-empty stacks (similar to our list $\mathcal{S}$ of stacks with non-marked heads). However, their approach does not suffice for computing $\mathcal{H}$ optimallty as in the worst case $n' = O(n^2) >> O(||P||)$.

Algorithm 3 MARK$\mathcal{H}(\mathcal{A}, \mathcal{L})$

```
foreach internal node v of A do in_H[v] ← 0      // Flag for membership in H
foreach leaf v of A do in_H[v] ← 1               // Leaves implicitly in H
in_H[root] ← 1                                   // Root implicitly in H

foreach leaf v of A do S_v ← ∅                   // Stack of exposed suffix
S ← ∅                                            // List of stacks with unmarked tops
foreach leaf v of A do in_S[v] ← 0               // Flag for membership of S_v in S

foreach node v ∈ A in DFS order do               // Compute all in_H[v]
    if internal node v first visited then
        foreach leaf x in L_v do                 // Expose v on stacks of L_v
            Push v on S_x
            if in_S[x] = 0 then                  // Add S_x to S if not present
                in_S[x] ← 1
                Add S_x to S
    if internal node v last visited then
        foreach leaf x in L_v do                 // Remove v from stacks of L_v
            Pop v from S_x
            if S_x ≠ ∅ and in_H[Top of S_x] = 0 then   // S_x eligible in S
                if in_S[x] = 0 then              // S_x not present in S
                    in_S[x] ← 1
                    Add S_x to S
            else                                 // S_x either empty or with marked top
                if in_S[x] = 1 then              // S_x present in S
                    in_S[x] ← 0
                    Remove S_x from S
    if leaf v visited then
        foreach S_x ∈ S do
            in_H[Top of S_x] ← 1                 // Mark ov(x, v)
            in_S[x] ← 0
            Remove S_x from S                    // Remove S_x with marked top from S

Return in_H
```

5 Conclusions

Genome assembly is one of the most prominent problems in Bioinformatics, and it traditionally relies on de Bruijn graphs or Overlap graphs, each having limitations of either loss of information or quadratic space requirements. Hierarchical Overlap Graphs provide a promising alternative that may result in better algorithms for genome assembly. The previous results on computing these graphs were not scalable (due to the quadratic time-bound) or required complicated data structures (segment trees). Moreover, computing HOG in optimal time and space was mentioned as an open problem in both the previous results [10, 20]. We present a simple algorithm that achieves the desired bounds, using only elementary data structures such as stacks and lists. At its core, we present an improved algorithm for a

special case of All Pairs Suffix Prefix problem. We hope our algorithm directly, or after further simplification, results in a greater adaptability of HOGs in developing better genome assembly algorithms.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

2 Dmitry Antipov, Anton I. Korobeynikov, Jeffrey S. McLean, and Pavel A. Pevzner. hybridspades: an algorithm for hybrid assembly of short and long reads. *Bioinform.*, 32(7):1009–1015, 2016.

3 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son K. Pham, Andrey D. Prjibelski, Alex Pyshkin, Alexander Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. Spades: A new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, 19(5):455–477, 2012.

4 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

5 Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. Bidirectional variable-order de bruijn graphs. *Int. J. Found. Comput. Sci.*, 29(8):1279–1295, 2018.

6 Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.

7 Christina Boucher, Alexander Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 383–392. IEEE, 2015.

8 Rodrigo Cánovas, Bastien Cazaux, and Eric Rivals. The compressed overlap index. *CoRR*, abs/1707.05613, 2017. `arXiv:1707.05613`.

9 Bastien Cazaux, Rodrigo Cánovas, and Eric Rivals. Shortest DNA cyclic cover in compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016*, pages 536–545. IEEE, 2016.

10 Bastien Cazaux and Eric Rivals. Hierarchical overlap graph. *Inf. Process. Lett.*, 155, 2020.

11 Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

12 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

13 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992.

14 Shahbaz Khan. Optimal construction of hierarchical overlap graphs. *CoRR*, abs/2102.02873, 2021. `arXiv:2102.02873`.

15 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

16 Jihyuk Lim and Kunsoo Park. A fast algorithm for the all-pairs suffix-prefix problem. *Theor. Comput. Sci.*, 698:14–24, 2017.

17 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(2):79–85, 2005. `doi:10.1093/bioinformatics/bti1114`.

18 Sergey Nurk, Dmitry Meleshko, Anton I. Korobeynikov, and Pavel A. Pevzner. metaspades: A new versatile de novo metagenomics assembler. In Mona Singh, editor, *Research in Computational Molecular Biology - 20th Annual Conference, RECOMB 2016, Santa Monica, CA, USA, April 17-21, 2016, Proceedings*, volume 9649 of *Lecture Notes in Computer Science*, page 258. Springer, 2016.

19 Sangsoo Park, Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. A linear time algorithm for constructing hierarchical overlap graphs. *CoRR*, abs/2102.12824, 2021 *(accepted for publishing at CPM 2021)*. `arXiv:2102.12824`.

20 Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. Efficient construction of hierarchical overlap graphs. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 277–290. Springer, 2020.

21 Hannu Peltola, Hans Söderlund, Jorma Tarhio, and Esko Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *IFIP Congress*, pages 59–64, 1983.

22 P. A. Pevzner. l-Tuple DNA sequencing: computer analysis. *Journal of Biomolecular Structure & Dynamics*, 7(1):63–73, 1989.

23 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–9753, 2001.

24 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinform.*, 26(12):367–373, 2010.

25 Z. Sweedyk. A 2½-approximation algorithm for shortest superstring. *SIAM J. Comput.*, 29(3):954–986, 1999.

26 William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix-prefix problem. *J. Discrete Algorithms*, 37:34–43, 2016.

27 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990.

28 Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821—829, May 2008. `doi:10.1101/gr.074492.107`.